Code 582

Flight Software Branch

FSW UNIT TEST STANDARD

Flight Software Branch – Code 582

Version 1.03 – 09/25/03

582-2000-002

Goddard Space Flight Center

Greenbelt, Maryland

National Aeronautics and
Space Administration

FORWARD AND UPDATE HISTORY

This standard defines the NASA Goddard Space Flight Center Flight Software Branch software library requirements for unit tests of flight software components, and gives guidance on unit test implementation.

Version	Date	Description	Affected Pages
1.0a	7 Sept 2000	initial	all
1.01	3 Sept 2003	Convert source to Microsoft Word, modify formatting	all
1.02	3 Sept 2003	Modify requirement for testing on target machine. Formatting & layout changes; new cover page	all
1.03	25 Sept 2003	Modify requirement 6 to describe waiver process	

CONTENTS

1 INTRODUCTION

Testing is an essential part of maintaining a software library. Users of code from the library must be able to trust the code. Having easy to run, complete unit tests contributes to this goal.

At the same time, having well tested library units allows projects to do better testing on their specific target hardware. Since they are not designing and writing code and unit tests from scratch, they have more time to run the unit tests on their specific hardware, in order to ferret out the last remaining compiler or target specific bugs.

A unit is defined as single interface file, together with the body files that implement the unit. In C, the interface file is a header file. In Ada, it is a package specification.

A unit test is defined to be a test that executes all of the code in a unit, with the unit not embedded in a larger system. The unit test proves that the unit executes correctly. Unit tests are always white-box; they take advantage of knowledge of how the unit is constructed. Unit tests may consist of more than one executable, if that makes them more understandable.

A unit test is distinct from a functional test, which proves that a unit meets its requirements as part of a larger system. Functional tests are always “black box”; they use no knowledge of how the unit is constructed - they can only use knowledge of the unit’s interface and requirements.

It may be necessary to break up large units into smaller ones in order to satisfy the requirement for full path coverage and boundary value testing. The large units then become components.

It is often the case that algorithm bugs are found during unit testing; the unit developer may discover an ambiguity in the algorithm definition, or it may become clear that the “known good” output is actually wrong. While it is always useful to find such bugs, this is not the primary goal of unit testing. Unit tests should show that the unit implements the algorithm, not that the algorithm is “correct”.

Another type of testing is “performance testing”, where a full system must meet some performance criteria, such as pointing a telescope with some accuracy. Unit testing does not test performance, except that it may be useful to measure execution times for various operations.

One issue not covered here is testing the effect of single event upsets, which can change arbitrary portions of the code and data.

2 REQUIREMENTS

1. Unit test shall clearly demonstrate that the unit is correctly implemented.

It must be possible, by reading the component document, the unit source code, the unit test source code, and the unit test output, to decide whether the unit source code correctly implements the design.

There must be sufficient design documentation on the unit test to make it clear what is being tested, and the general test approach.

2. Unit test shall execute every statement in the unit, including all branches of conditional statements.

This is sometimes called “full path coverage”. The point is to never execute code for the first time in flight.

We do not require covering all combinations of paths, since that is often too cumbersome.

For example, consider the following code:

```
if (a=1)
{
   if (b=1)
      ... /* line 1 */
   else
      ... /* line 2 */

   if (c=1)
      ... /* line 3 */
   else
      ... /* line 4 */
}

... /* line 5 */
```

This code only requires 2 tests for "full path coverage". Lines 1, 3, 5 are executed by the case a=1, b=1, c=1, and lines 2, 4 are executed by the case a=1, b=2, c=2.

Covering all combinations of paths would mean requiring the additional cases a=1, b=1, c=2 (to get the total path line 1, line 4, line 5) and a=1, b=2, c=1 (to get line 2, line 3, line 5). This standard does not require all combinations of paths.

The test documentation must make it clear what paths are being covered by each test case. This can be done with comments in the unit source that label paths, paired with comments in the unit test code. Line numbers are not good as labels, since maintenance activities can easily change a line number.

3. Each conditional branch in the unit shall be executed with data at each side of the boundary of the condition, and with data away from the boundary on each side.

This is sometimes called “boundary value testing”. For example, if there is a condition “n < 3”, with n an integer, the largest value of n that will yield true is 2, and the smallest value that will yield false is 3. Thus the data at the boundary is n = 2 and n = 3, and data away from the boundary could be n = 0 and n = 5. This helps show whether the “<” should instead be “<=”.

The test documentation must make it clear which conditional branch is being tested. This can be done with labeling comments in the unit source code, paired with comments in the unit test code.

4. All operations which might cause erroneous execution, such as divide by zero, taking the square root of a negative number, etc., shall be either proved impossible or tested explicitly.

The goal is to have the error occur for the first time in testing, not on flight.

If the erroneous execution can only happen for certain combinations of paths, those paths must be tested or proved impossible.

For example, consider the following code:

```
int x;
int y;

if (b=1)
   x = 1; /* line 1 */
else
   x = 0; /* line 2 */

if (c=1)
   ... /* line 3 */
else
   y = 2 / x; /* line 4 */
```

Since line 4 may be erroneous depending on whether line 1 or line 2 is taken, these total paths must be tested explicitly.

5. All parameters and inputs to subprograms shall be tested with nominal values, and with values at the extremes allowed by the algorithm, compiler, or CPU.

In some cases, particularly for floating point values, there is no identifiable extreme value, and this requirement may be relaxed.

Testing with nominal values makes it easier to see if the results are reasonable.

Testing with extreme values finds overflow and underflow errors.

Sometimes units have parameters stored in tables, as opposed to being passed directly in function calls. The effects of these table parameters must be tested in the same way as all other parameters.

Sometimes certain combinations of parameters can reveal bugs; one parameter being zero is not a problem unless another one is also. All such cases must be identified and tested.

6. Unit tests shall not require any changes to the module source code. If a preprocessor is used, exactly the same preprocessor values must be set for the final release and unit test builds.

Otherwise you are not testing the same code. The compiler may optimize things differently if debug code is present in the module code; this may hide a compiler bug.

Debug code is often used to print intermediate results to make the unit test output more understandable. If this is necessary, provide storage for the intermediate results in a place the unit test code can access. It will often be desirable to provide the same intermediate results in telemetry, so a problem can be diagnosed in flight. Using a debugger to access the intermediate results is not a good idea, since it is much less portable to other target systems.

Sometimes stubs or alternate modules are required for parts of the full system that the unit under test interfaces with. These alternates are part of the unit test, not part of the unit, and as such can contain code specifically for performing the unit test, including reading input files and writing output files.

Software that writes directly to hardware is always difficult to unit test. The full system should be structured so that there is a very thin layer that writes to hardware; all other software can be tested with an alternate body for this hardware layer, that writes to an output file to show that the hardware write functions are called. Then only the thin hardware layer needs to be tested with the real hardware.

In some situations, it will only be possible to test some functionality in the unit by modifying the unit source. This requires a waiver from the product development lead, and must be done using preprocessor settings (not manual editing), following the language-specific coding standards for such cases.

7. It shall be possible for the unit tests to be compiled with the target compiler, using the same configuration switches as the final target build, and run on the target hardware (or equivalent test hardware). There shall be documentation of which targets the unit has been tested on.

The purpose of using the target compiler on the target hardware is to find compiler and target hardware bugs. If this is not possible, a unit test configuration as close as possible to this is desired.

The branch supports multiple targets, so we have to know which ones this unit test has actually run on.

Different projects may require different configuration switch settings; the documentation must include this information, and the tests should be repeated with the different configurations. The goal is to allow the projects to reuse the library code with minimal re-testing effort.

Preliminary runs of the unit test on other hardware, or with other compilers or compiler configuration switches (for example to enable saving debug information), are permitted during unit test development. This allows finding logic, design or requirement bugs under a friendlier debugging environment.

On some projects, access to the target hardware is a scarce resource, so unit tests may only be run on the target hardware if a target compiler or hardware bug is suspected.

8. Unit tests shall be repeatable, producing identical results on each run.

If possible, a single output file should be generated, that can be saved and differenced against subsequent results. When an output file is generated, it should contain all inputs, parameters, and outputs for each test case, clearly labeled. If a different format is needed to facilitate plotting or comparing with known good output, a second output file should be generated. In some cases, multiple output files will make more sense.

Having an output file to diff makes it easy to tell if the unit test has passed or not.

Note that ANSI C does not define how many digits are in the exponent in 'e' format; where possible, use 'f' format instead.

In many cases, the requirements documentation for the unit will contain a file of known good results; a test output file should be written to reproduce that file, so it is easy to tell if the unit passes the test. Or, have the unit test read the known good file, subtract it from the unit output, and define the pass/fail criteria based on the average error or other appropriate measure.

In other cases, the unit will be the first implementation of the algorithm; then the algorithm author must approve the unit test output file as being correct.

Using a plot as the definition of expected results is usually not a good idea, since it is labor-intensive and somewhat subjective to "compare plots" to see if the unit passes the test. Plots are very useful while debugging, but they are not useful in the final unit test. The ability to produce a suitable plot file should be maintained in the final unit test, to facilitate debugging changes to the algorithm.

9. Unit tests shall run without user interaction, as much as possible.

Having tests run automatically means they will be run more often, since it is easy. A desirable scenario is to run all unit tests after each major build; having an automated unit test makes this much easier.

It is highly recommended that all unit tests be run from a single command (the main makefile, for example).

10. If input data files are used by the unit test, they shall be treated as source code for the purpose of configuration management. Such input data files shall be in human readable form; if this is not possible, a separate tool shall be provided to generate the actual test input files from some human readable form.

It is desirable to allow comments in input files, to document the general purpose of that particular test, and to label the various input values to make it easy to edit them. One simple commenting convention is to ignore lines at the start of the file that begin with a suitable comment character.

Note that the ut69r systems do not support input files; unit tests that must run on a ut69r must have all input data coded in the executable.

11. Distinct elements of input vectors and matrices shall have distinct values.

This makes it possible to catch indexing errors.

12. When the order of inputs to an operation matters, the inputs shall have distinct values.

This makes it possible to catch order errors, such as for matrix multiply or cross products.

3 GUIDELINES

It is often useful to have a library of unit test support code, providing standard output facilities or stubs. Such code should be treated as part of the library, rather than as part of each unit test.

When parameters are used in equations, it is often useful to run separate tests with the parameters set to 0, 1, and some nominal value from a real scenario. Using 0 may reveal divide-by-zero problems; using 1 makes it easy to compute the equation, showing that the output is simply related to the input; using a nominal value makes it easy to see if the output is reasonable.

Sometimes units depend on a particular operational sequence for correct operation. For example, a filter must be initialized before it is used. Ideally, the unit can detect that it is not initialized, and report an error in some way. If so, this case must be tested. However, in a real time system it is often preferable to just assume the initialization is done, rather than wasting time and code space doing the check each time. In these cases, the test documentation must clearly state this operational requirement, as justification for why that particular path is not being tested.

If it seems to be hard to construct test cases that adequately cover the unit, it may be that the unit is badly designed. One of the requirements for a well-designed unit is that it be testable, and that the test be understandable.